MONITORING SOFTWARE DEVELOPMENT THROUGH DYNAMIC VARIABLES

Carl W. Doerflinger
Victor R. Basili

University of Maryland
Dept. of Computer Science
College Park, MD 20742
(301) 454-2002

Abstract

This paper describes research conducted by the Software Engineering Laboratory (SEL) on the use of dynamic variables as a tool to monitor software development. The intent of the project is to identify project independent measures which may be used in a management tool for monitoring software development. This study examines several FORTRAN projects with similar profiles. The staff was experienced in developing these types of projects. The projects developed serve similar functions. Because these projects are similar we believe some underlying relationships exist that are invariant between the projects. These relationships, once well defined, may be used to compare the development of different projects to determine whether they are evolving the same way previous projects in this environment evolved.

Overview

The Software Engineering Laboratory (SEL) is a joint effort between the National Aeronautics and Space Administration (NASA), the Computer Sciences Corporation (CSC), and the University of Maryland established to study the software development process. To this end, data has been collected for the last six years. The data was from attitude determination and control software developed by CSC, in FORTRAN, for NASA. Additional information on the SEL, the data collection effort, and some of the studies that have been made may be found in papers from the Software Engineering Laboratory Series published by the SEL[1,2,3].

This research was supported by the National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center.

The interest in the software development process is motivated by a desire to predict costs and quality of projects being planned and developed. For several years, studies have examined the relationships between variables such as effort, size, lines of code, and documentation[4,5]. These studies, for the most part, used data collected at the end of past projects to predict the behavior of similar projects in the future. In 1981 the SEL concluded that many of these factors were too dependent on the environment to be useful for the models that had been developed[6]. Any model which attempts to trace these relationships should therefore be calibrated to the environment being examined. The meta-model proposed by the SEL is designed for such flexibility[6].

Another way to isolate out the environment dependent factors is by comparing two internal factors of a project, thus ignoring all outside influences. One approach that is used to monitor software development examines the time gap between the initial report of software problems and the complete resolution of the problem[7]. Comparing two variables is useful because it also accentuates problem areas as they develop, providing relative information rather than absolute information. Relative information is useful to the project manager because it accentuates trends as the project develops. If project environments are similar, then similar values should be expected. Because the project environments in the SEL are similar, it was felt that this approach could be further extended to provide managers with information about how a set of variables over the course of a project differed from the same set of variables on other projects (baselines). The managers could be alerted to potential problems and use other variable data and project knowledge to determine whether the project was in trouble.

This methodology is flexible enough to respond to changing needs. Every time a project is completed the measures collected during its development may be added in to calculate a new baseline. In this changes in the environment, as they occur.

Baselines might also be developed to reflect different attributes. For instance, several projects which had good productivity might be grouped to form a productivity baseline. Once baselines are established, projects in progress may be compared against them. All measures falling outside the predetermined tolerance range are interpreted by the manager.

Methodology

The implementation of this methodology is dependent on two factors. The first factor is the availability of measures that are project independent and can also be collected throughout a project's development. Variables like programmer hours and number of computer runs are project dependent. By comparing these variables against each other a set of relative measures may be generated which is project independent. For instance, the number of software changes may vary from project to project. The project dependent features shared by each variable will cancel out when the ratio of software changes per computer run is taken. The resulting relative measure is project independent.

The second factor is the need for fixed time intervals common to all projects. To normalize for time, project milestones were used. The time into a project might be twenty percent into coding instead of ten weeks into the project, for instance.

When computing the baselines one other factor was considered. At any given interval during development a variable may measure either the total number of events that have occurred from the beginning of development (cumulative) or the number of of events that have occurred since the last measured interval (discrete). Since these approaches may convey different information it was felt that they both should be used.

For simplicity, the baseline for each relative measure was defined as the average and standard deviation computed for the measure at predetermined intervals. A project's progress may now be charted by the software manager. At each interval in a projects development the relative measures are compared with their respective baseline. Any measures outside a standard deviation are flagged. These measures are then interpreted by the project manager to determine how the project is progressing. A flagged measure may indicate a project is developing exceptionally well or it may indicate a problem has been encountered.

The interpretation of a set of flagged measures is a three step process. First, the manager must determine the possible interpretations for each flagged relative measure using lists of possible interpretations developed and verified based on past projects.

Second, the union of the lists of possible interpretations of each flagged measure must be taken. The list formed by this union contains all the possible interpretations ordered using the number of times each interpretation is repeated in the different lists. The larger the number of overlaps a possible interpretation has, the greater the probability it is the correct interpretation.

Third, the manager must analyze the combined list and determine if a problem exists. Interpretations with an equal number of overlaps all have an equal probability of being the correct interpretation. If none of the possible interpretations for a given relative measure overlap then the relative measure should be considered separately.

When analyzing the interpretations, three pieces of information must be considered; the measurements, the point in development, and the managers knowledge of the project. A relative measure may indicate different things depending on the stage of development. For instance, a large amount of computer time per computer run early in the project may indicate not enough unit testing is being done. Personal knowledge may also give valuable insight.

A fundamental assumption for using this methodology is that similar type projects evolve similarly. If a different type of project was compared to this database, the manager would have to decide whether the baselines were applicable. Depending on the type of differences, the established baselines may or may not be of any value.

EXAMPLE 1:

Forty percent into coding a software manager finds that the lines of source code per software change is higher than normal. A list previously developed is examined to determine what the relative measure might indicate. The possible interpretations for a large number of lines of source code per software change might be:

- good code
- easily developed code
- influx of transported code
- near build or milestone date
- computer problems
- poor testing approach

If this were the only flagged measure the manager would then investigate each of the possibilities. If the value for the measure is close to the norm less concern is needed than if the value is further away.

If in addition to lines of source code per software change the number of computer runs per software change was higher than normal, the manager would also examine this measure. The possible interpretations for a large number of computer runs per software change might be:

- good code
- lots of testing
- change backlog
- poor testing approach

The union of the possible interpretations of these two measures indicates that the strongest possible interpretations are 1) good code and 2) a poor testing approach. The number of possibilities to investigate is smaller because these are the only measures which overlap. The manager must now examine the testing plan and decide whether either of these interpretations reflect what is actually occurring in the project. If these two possible interpretations do not reflect what is happening on the project, the manager would then examine the other interpretations.

Baseline Development

To develop a baseline one must first have variables whose measurements were taken weekly for several projects. Five variables in the SEL database were used. The lines of source code, number of software changes, and number of computer runs were collected on the growth history form. The amount of computer time and programmer hours were collected on the resource summary form. Measurement of these variables started near the beginning of coding. In this study, nine separate projects were examined whose development was documented, with sufficient data, in the SEL database. The projects ranged in size from 51-112K lines of source code with an average of 75K. No examination was done for the requirements or design phases.

Once the variables were chosen the average and standard deviation was computed for each baseline. Some baselines suffered from limited data points during the beginning of the coding phase. A couple of the projects, in which problems were known to have existed, were flagged as soon as data on these projects appeared, but this was fifty percent of the way into coding. It is not known how much earlier they would have appeared, if data existed at the early intervals.

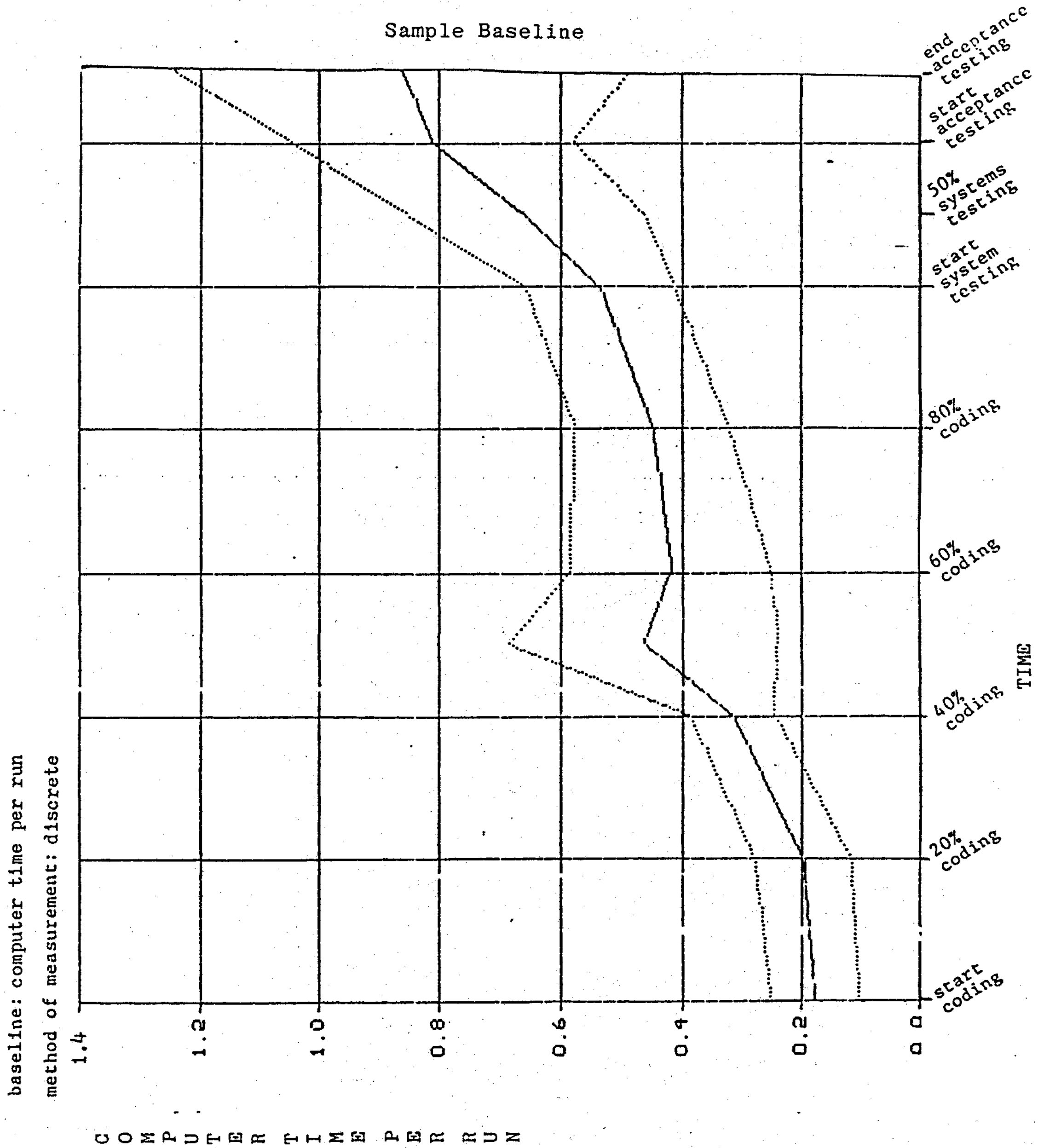
Interpretation of Relative Measures

Once a set of baselines are established new projects may be compared to them and potential problems flagged. To interpret these flagged relative measures a list should be developed with each measures possible interpretations. Each list must consider the possible interpretations of the relative measure when it is either above normal or below normal. What each component variable actually measures should also be considered when the different lists are developed.

A list was developed with possible interpretations for each relative measure being examined in the context of the SEL environment. In another environment the interpretation of these measures might be different. These lists are subdivided into two categories; above and below normal. The above normal category contains possible interpretations for the relative measure when it is outside one standard deviation from the average in the positive direction. The below normal category refers to interpretations when the measure is outside one standard deviation from the mean in the negative direction.

One of the reasons this methodology works is because of the implicit interdependencies between different relative measures. To show these interdependencies more explicitly a cross reference chart has also been provided for each interpretation to indicate other relative measures that can have the same interpretation. A number in the cross reference section indicates the list number of a relative measure that can have the same interpretation. The position of the list number in the 4-quadrant cross reference section indicates whether both interpretations are found with above normal values, both with below normal values, or one with above and the other with below normal values.

With these lists a set of flagged relative measures may be evaluated. When a relative measure is flagged, its associated list is examined for possible interpretations. Overlaps of this list with the lists of other flagged relative


Sample Baseline
baseline: computer time per run
method of measurement: discrete
COMPUTER TIME PER RUN
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
TIME
start coding
20% coding
40% coding
60% coding
80% coding
start system testing
50% systems testing
start acceptance testing
end acceptance testing

Relative Measures Examined:

List 1 - Computer Runs per Line of Source Code
List 2 - Computer Time per Line of Source Code
List 3 - Software Changes per Line of Source Code
List 4 - Programmer Hours per Line of Source Code
List 5 - Computer Time per Computer Run
List 6 - Software Changes per Computer Run
List 7 - Programmer Hours per Computer Run
List 8 - Computer Time per Software Change
List 9 - Programmer Hours per Software Change

List 1: Computer Runs per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-low productivity	2 4	
	-high complexity	2 4 7 8 9	
	-lots of testing	2	6 7
	-removal of code (testing or transported)	2 3 4	
	-bad specifications	2 3 4	
below normal			
	-influx of transported code		2 3 4
	-near build or milestone date	6	2 3 4 8 9
	-little on line testing being done		2
	-little executable code being developed		2
	-computer problems		3

List 2: Computer Time per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-high complexity	1 4 7 8 9	
	-low productivity	1 4	
	-bad specifications	1 3 4	
	-lots of testing	1	6 7
	-unit testing being done	8	5
	-code being removed (testing or transported)	1 3 4	
below normal			
	-influx of transported code		1 3 4
	-near build or milestone date	6	1 3 4 8 9
	-little on line testing being done		1
	-code error prone	3 4 5 6	7 8 9
	-little executable code being written		1

List 3: Software Changes per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good testing	6	8 9
	-error prone code	4 5 6	2 7 8 9
	-bad specifications	1 2 4	
	-code being removed (testing or transported)	1 2 4	
below normal			
	-influx of transported code		1 2 4
	-near build or milestone date	6	1 2 4 7 8
	-good code	8 9	6
	-poor testing program	8 9	6
	-change backlog		6
	-low complexity		4
	-computer problems		1

List 4: Programmer Hours per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-high complexity	1 2 7 8 9	
	-error prone code	3 5 6	2 7 8 9
	-bad specifications	1 2 3	
	-code being removed (testing or transported)	1 2 3	
	-changes hard to isolate	7 8 9	
	-changes hard to make	7 9	
	-low productivity	1 2	
below normal			
	-influx of transported code		1 2 3
	-near build or milestone date	6	1 2 3 8 9
	-low complexity		3

List 5: Computer Time per Computer Run

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-system & integration testing started early	6	
	-error prone code	3 4 6	2 7 8 9
	-compute bound algorithms being tested	8	
below normal			
	-unit testing going on	2 8	
	-easy errors being found		7 9

List 6: Software Changes per Computer Run

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good testing	3	8 9
	-system & integration testing started early	5	
	-error prone code	3 4 5	2 7 8 9
	-near build or milestone date		1 2 3 4 8 9
below normal			
	-good code	3 8 9	
	-lots of testing	1 2	7
	-poor testing program	3 8 9	
	-change backlog	3	

List 7: Programmer Hours per Computer Run

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-high complexity	1 2 4 8 9	
	-modifications being made to recently transported code		9
	-changes hard to isolate	4 8 9	
	-changes hard to make	4 9	
below normal			
	-easy errors being fixed		5 9
	-error prone code	3 4 5 6	2 8 9
	-lots of testing	1 2	6

List 8: Computer Time per Software Change

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good code	3 9	6
	-poor testing program	3 9	6
	-high complexity	1 2 4 7 9	
	-changes hard to isolate	4 7 9	
	-unit testing	2	5
	-compute bound algorithms being tested	5	
below normal			
	-near build or milestone date	6	1 2 3 4 9
	-good testing	3 6	9
	-error prone code	3 4 5 6	2 7 9

List 9: Programmer Hours per Software Change

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good code	3 8	6
	-poor testing program	3 8	6
	-changes hard to isolate	4 7 8	
	-changes hard to make	4 7	
below normal			
	-good testing	3 6	8
	-near build or milestone date	6	1 2 3 4 8
	-easy changes		5 7
	-transported code being modified	7	
	-error prone code	3 4 5 6	2 7 8

measures form the new list of what these relative measures together might indicate. The more overlaps a particular interpretation has, the greater the chance it is the correct interpretation. Interpretations with the same number of overlaps must be considered equally. The more relative measures flagged the more serious the problem may be. It is up to the manager to determine whether the deviation is good or bad.

Monitoring a Software Project's Development

Once the baselines have been developed and the lists of possible interpretations have been put together a software manager may monitor the actual development of a project. Example 1 demonstrated how a single interval may be interpreted. The following discussion will trace the development of an actual project. During the actual use of this methodology, influence would be exerted to correct problems as soon as they are identified. With this study, we must be content to study a projects evolution, without hindrance, and see at what points problems could of been detected.

Project twenty* was chosen for this examination because data existed throughout the projects development. In most respects project twenty was an average project. The project did have a lower than normal productivity rate. The lower rate may be partially explained by the fact the management was less experienced when compared to other projects. The project also suffered from some delayed staffing. Changes in staffing will be noted when the different time intervals are discussed.

The tables on the following page show which relative measures were flagged when project twenty was compared to the baselines for each stage of development. The numerical values represent how many standard deviations each flagged relative measure was from the baseline. The baseline for each relative measure was calculated using all nine projects.

Start of Coding:

At the start of coding only one relative measure is flagged. The smaller than normal number of software changes per line of source code using the discrete approach reflects work done during the design phase. The lists designed in the previous section were directed towards code production and testing and do not apply to this time interval when using the discrete approach. This measure may indicate good specifications or lots of PDL being generated. The manager might want to examine this measure later if it constantly repeated. Since it is the only measure flagged at this time it will be ignored.

---

* The numbering convention used is an extension of the one first used by Bailey and Basili[6].

project: 20

method of measurement: cumulative

number of standard deviations from norm										relative measures
start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	
									1.3	>1 SD programmer hours/lines of source
						1.1				>1 SD runs/lines of source
						1.8	1.5	1.2		>1 SD computer time/lines of source
		1.1	1.2	1.1		1.1				<1 SD programmer hours/run

method of measurement: discrete

number of standard deviations from norm										relative measures
start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	
	1.0	1.1	1.8			1.5		2.0	2.4	>1 SD programmer hours/lines of source
	1.2		1.8			1.8		1.7		>1 SD runs/lines of source
1.1										<1 SD changes/lines of source
	1.1		1.1			2.0		2.0	2.4	>1 SD changes/lines of source
	1.2	1.3	1.7			2.1		2.0		>1 SD computer time/lines of source
	1.2									<1 SD programmer hours/run
						1.2				>1 SD computer time/change

20% Coding:

The flagged relative measures found using the discrete approach at this point represent the work done from the start of coding until twenty percent of the way through coding. The list of possible interpretations for the flagged relative measures, generated from the lists made previously for the individual relative measure, would look like:

# overlaps	interpretation
3	bad specifications
3	code removed
2	low productivity
2	high complexity
2	error prone code
1	lots of testing
1	good testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	easy errors being found

The strongest interpretations are bad specifications and code being removed. If the actual history is examined one finds that during this period there were a lot of specifications being changed. This resulted in code which was to be modified being discarded and new code being written. During the early period lots of PDL was being produced but very little new executable code. The list of possible interpretations does show that low productivity is also a strong possibility.

40% Coding:

The flagged relative measures which appear using the cumulative approach, from this time period on, are stronger indicators than the ones used in the first couple of intervals because the average is computed using more data points. The use of the discrete approach for the interval of twenty to forty percent is still dependent on three data points. The list of possible interpretations for this time period is:

# overlaps	interpretation
1	low productivity
1	high complexity
1	error prone code
1	bad specifications
1	code being removed
	changes hard to isolate
	changes hard to make
	lots of testing
	unit testing being done
	good testing
	easy errors

The number of possibilities is larger with this set of possible interpretations. Five interpretations are slightly stronger than the others. During the actual development, the first release of the project was made. The amount of code actually written was also lower than normal during this period. The use of the discrete approach gives a stronger feeling that code is not being written. Transported code tends to be installed in large blocks which can be isolated using the discrete approach.

50% Coding:

The relative measures flagged during this period are the same as the ones flagged at the twenty percent coding interval. The deviation from the norm for this interval is larger. The larger deviation may indicate a more serious problem. The problem may of been just as serious earlier but without the extra data points, that are now available, it could not be determined. The possible interpretations may be taken from the list developed earlier. Bad specifications and code removal were not factors during this period. The next three highest priority interpretations were; high complexity, error prone code, and low productivity. In addition to this the manager should be concerned with the continued appearance of the relative measure, programmer hours per computer run, as seen using the cumulative approach. This may indicate a lot of testing going on. This in conjunction with error prone code as a possible interpretation may indicate trouble. During actual development this period was spent developing code for the second release. The project manager felt that code was still not being developed quickly enough during this period.

60% Coding:

Only one relative measure is shown at this interval. The number of programmer hours per computer run using the cumulative approach is lower than normal for the third consecutive time. This should concern the manager because when examining the list for this measure one finds:

error prone code
lots of testing
easy errors being fixed

Since the occurrence of this measure is persistent it may indicate that the problem was corrected but not enough effort was expended to completely compensate for the past problems. It might also indicate the problem still exists. During the actual project it was found that while a lot of code was written, it had not been throughly tested. Release two was made during this period which could explain a heavy test load. Two additional staff members were added to the project during this phase to aid in coding and testing.

80% Coding:

The eighty percent coding interval does not show any measures outside the normal bounds. The addition of two staff members during the sixty percent coding phase, as well as the addition of a senior staff member during this phase, appears to have adjusted the project back along the lines of normal development. To fully compensate for the earlier problems one might expect some of the measures to swing in the other direction away from the average. The fact this over correction did not occur might explain the problems encountered in the next section.

Start of System and Integration Testing:

The flagged relative measures at this time period reflect the build up of effort for the third and final release. The list of possible interpretations for the collective set of flagged measures looks like:

# overlaps	interpretation
3	high complexity
3	bad specifications
3	code being removed
2	error prone code
2	low productivity
2	lots of testing
1	changes hard to isolate
1	unit testing being done
1	good code
1	poor testing
	changes hard to make
	good testing
	compute bound algorithms being run
	easy errors being fixed

Since the code did have a past history of poor testing an unusually large build up of testing should be expected. The two interpretations that apply most to this situation are lots of testing and error prone code.

50% System and Integration Testing:

Only one relative measure is flagged at this interval. This measure was flagged using the cumulative approach. An examination of the measure at the previous interval shows a very high value. A slow drop off from this high measure is to be expected when using the cumulative approach. An examination of possible interpretations that would apply for this period of development include:

high complexity
lots of testing
unit testing being done
testing code being removed

A lot of testing is certainly indicated by past history.

Start Acceptance Testing:

The relative measures flagged at this interval reflects the build up in testing before the start of acceptance testing. The list of possible interpretations looks like:

# overlaps	interpretation
3	bad specifications
3	code being removed
2	high complexity
2	low productivity
1	error prone code
1	lots of testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	good testing

Since little code was being developed during the testing period, a large amount of testing with errors being found is the most reasonable interpretation of these flagged measures. The early history of poor testing may be seen here with errors being uncovered late.

End Acceptance Testing:

The two flagged relative measures at the end of acceptance testing reflect the clean up effort being made on the code. An average amount of computer time and an average number of computer runs indicates that the acceptance testing is going well. The project was behind schedule due to the earlier problems encountered. Clean up was done during the acceptance testing phase in an attempt to get the project out the door as soon as possible.

As seen in this example, the problems that occur during a projects development are reflected in the values calculated for the relative measures. The methodology preposed can be used to monitor projects. The number of possible interpretations increases with each new flagged relative measure. The ordering of the measures by the number of overlaps provides an easy method of sorting the possible interpretations by priority. Another method of sorting the possible interpretations could include a factor that considers both the number of overlaps and the probability of a given interpretation being the cause at a given interval. The weighting of interpretations for a given interval could be calculated using the pattern of occurrence of the different interpretations which have appeared during the same interval in past projects.

An Alternate Approach

Flagged relative measures might also be interpreted using a decision support system. The data for the various relative measures would be stored in a knowledge base along with a set of production rules. To evaluate a project the values for each relative measure would be entered into the system. The knowledge base would compare the relative measures to their respective baselines, determine which relative measures were outside the norm, and interpret these relative measures using the production rules. A list of possible interpretations ordered by probability would be generated as a result.

The difference between a decision support system and the approach presented in this paper is the method of interpreting the flagged relative measures. Each production rule in the decision support system is the logical disjunction of several flagged measures which yields a given interpretation. Each production rule is assigned a confidence rating which is then used to rate the possible interpretations. The lists for the relative measures provided earlier in the paper may be easily converted to production rules using the cross reference section. To develop the production rules for an interpretation one must generate the various combinations of relative measures which might reasonably imply the interpretation. Some relative measures may not imply a particular interpretation unless they are found in conjunction with another relative measure. Once the production rules are known and a knowledge base constructed a decision support system may be built. For an example of a domain independent decision support system see Reggia and Perricone[8].

Summary

The methodology presented in this paper showed that invariant relationships exist for similar projects. New projects may be compared to the baselines of these invariant relationships to determine when projects are getting off track.

The ability of the manager to interpret the measures that fall outside the norm is dependent on the amount of information the underlying variables convey. The manager must decide what attributes are to be measured (e.g. productivity) and pick variables that are closely related to them and are also measurable throughout the project. As an example, a variable like lines of code may be too general when measuring productivity. Measuring the newly developed code, either source code or executable code, would be more informative since these variables are more directly related to effort. How applicable an interpretation is for the period currently being examined should also be considered when ordering the list. The variables the manager finally decides on are then combined to form relative measures.

One method of interpreting a relative measure is by associating lists of possible interpretations with it. When a relative measure appears outside the norm, the list of possible interpretations is considered. If more than one relative measure is outside the norm the lists are combined. The more times a possible interpretation is repeated in the lists, the greater the probability it is the cause. How applicable an interpretation is for the period being examined should also be considered when ordering the list. The manager must investigate the suggested causes to determine the real one.

Conclusion

The ability to monitor a projects development and detect problems as they develop may be feasible. The methodology proposed showed favorable results when examining a past case.

The use of baselines and lists of interpretations for comparing projects provides an easy method for monitoring software development. Both the baselines and the lists of interpretations may be updated as new projects are developed. As more knowledge is gleaned the accuracy of this system should improve and provide a valuable tool for the manager.

Acknowledgements

The authors would like to thank Dr. Jerry Page of Computer Sciences Corporation and Frank McGarry of NASA/Goddard Space Flight Center for their insight and advice.

References

[1] Card, David, Frank McGarry, Jerry Page, Suellen Eslinger, and Victor Basili, The Software Engineering Laboratory, SEL-81-104, Software Engineering Laboratory Series, Goddard Space Flight Center, February 1982.

[2] Church, Victor, David Card, Frank McGarry, Jerry Page, and Victor Basili, Guide To Data Collection, SEL-81-101, Software Engineering Laboratory Series, Goddard Space Flight Center, August 1982.

[3] SEL,, Collected Software Engineering Papers: Volume 1, SEL-82-004, Software Engineering Laboratory Series, Goddard Space Flight Center, July 1982.

[4] Walston, C. E. and C. P. Felix, A Method of Programming Measurement and Estimation, IBM Systems Journal, January 1977.

[5] Basili, Victor R. and Karl Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, Journal of Systems and Software, 1981.

[6] Bailey, John W. and Victor R. Basili, A Meta-Model for Software Development Resource Expenditures, Proceedings, Fifth International Conference on Software Engineering, September 1981.

[7] The Role of Measurements in Programming Technology, Lecture presented at University of Maryland, November 15, 1982.

[8] Reggia, James and Barry Perricone, KMS Manual, TR-1136, Department of Mathematics, University of Maryland Baltimore County, January 1982.

[9] Minsky, M. L., A Framework for the Representation of Knowledge, The Psychology of Computer Vision, pp. 211-280, McGraw Hill, New York, 1975.